UDACITY

‹ Return to Classroom DISCUSS ON STUDENT HUB

DNN Speech Recognizer

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Congratulations on finishing the VUI Project!

STEP 2: Model 0: RNN

The submission trained the model for at least 20 epochs, and none of the loss values in `model_0.pickle` are undefined. The trained weights for the model specified in `simple_rnn_model` are stored in `model_0.h5`.

This simple model doesn't fit the data very well and thus the loss is very high.

STEP 2: Model 1: RNN + TimeDistributed Dense

The submission includes a `sample_models.py` file with a completed `rnn_model` module containing the correct architecture.

The submission trained the model for at least 20 epochs, and none of the loss values in `model_1.pickle` are undefined. The trained weights for the model specified in `rnn_model` are stored in `model_1.h5`.

Adding batch normalization and a time distributed layer improves the loss by ~6x! Try different units here, SimpleRNN, LSTM, and GRU to see how their performance differs.

STEP 2: Model 2: CNN + RNN + TimeDistributed Dense

The submission includes a `sample_models.py` file with a completed `cnn_rnn_model` module containing the correct architecture.

The submission trained the model for at least 20 epochs, and none of the loss values in `model_2.pickle` are undefined. The trained weights for the model specified in `cnn_rnn_model` are stored in `model_2.h5`.

These models are very powerful but have a tendency to severely overfit the data. You can add dropout to combat this.

STEP 2: Model 3: Deeper RNN + TimeDistributed Dense

The submission includes a `sample_models.py` file with a completed `deep_rnn_model` module containing the correct architecture.

The submission trained the model for at least 20 epochs, and none of the loss values in `model_3.pickle` are undefined. The trained weights for the model specified in `deep_rnn_model` are stored in `model_3.h5`.

Adding additional layers allows your network to capture more complex sequence representations, but also makes it more prone to overfitting. You can add dropout to combat this.

STEP 2: Model 4: Bidirectional RNN + TimeDistributed Dense

The submission includes a `sample_models.py` file with a completed `bidirectional_rnn_model` module

containing the correct architecture.

The submission trained the model for at least 20 epochs, and none of the loss values in `model_4.pickle` are undefined. The trained weights for the model specified in `bidirectional_rnn_model` are stored in `model_4.h5`.

These models tend to converge quickly. They take advantage of future information through the forward and backward processing of data.

STEP 2: Compare the Models

The submission includes a detailed analysis of why different models might perform better than others.

Very impressive job to try all kinds of combination and get a very deep understanding on the training progress among different models.
Very well done. You answer show your deep understanding on different model layers.

This is a pretty good analysis of each individual model. An improvement would be to explain why different models perform better than others. For example, what is it about the nature of a CNN on this problem that leads to extreme overfitting?

Additionally, you can look at things such as overfitting, the number of parameters that need to be tuned, and the training time for each individual model and compare these in a table.

STEP 2: Final Model

The submission trained the model for at least 20 epochs, and none of the loss values in `model_end.pickle` are undefined. The trained weights for the model specified in `final_model` are stored in `model_end.h5`.

Interesting model. It performs pretty well, and if you used more epochs I bet the loss would decrease even further.

However, you are overfitting by a little, and thus would likely see improved results if you were more aggressive with dropout and recurrent dropout.

Meanwhile, I think running out of GPU also makes your training a little painful. Sorry about that.

Here I listed a model that works well with me. It is quite similar to yours, but you can still talk a look :)

```
def final_model(input_dim, filters, kernel_size, conv_stride,
    conv_border_mode, units, dropout, output_dim=29):
    """ Build a deep network for speech
    """
    # Main acoustic input

    input_data = Input(name='the_input', shape=(None, input_dim))
    # TODO: Specify the layers in your network
    conv_1d = Conv1D(filters, kernel_size,
                     strides=conv_stride,
                     padding=conv_border_mode,
                     activation='relu',
                     name='conv1d')(input_data)
    # Add batch normalization
    bn_cnn = BatchNormalization(name='bn_conv_1d')(conv_1d)
    # Add a recurrent layer
    simp_rnn = SimpleRNN(units, activation='relu',
        return_sequences=True, implementation=2, dropout=dropout, name='rnn')
(bn_cnn)
    # Add batch normalization
    bn_rnn = BatchNormalization()(simp_rnn)
    # Add bidirectional recurrent layer
    bidir_rnn1 = Bidirectional(GRU(units, activation='relu',
        return_sequences=True, implementation=2))(bn_rnn)
    bidir_rnn2 = Bidirectional(GRU(units, activation='relu',
        return_sequences=True, implementation=2))(bidir_rnn1)
    # Add a TimeDistributed(Dense(output_dim)) layer
    time_dense = TimeDistributed(Dense(output_dim))(bidir_rnn2)
    # TODO: Add softmax activation layer
    y_pred = Activation('softmax', name='softmax')(time_dense)
    # Specify the model
    model = Model(inputs=input_data, outputs=y_pred)
    # TODO: Specify model.output_length
    model.output_length = lambda x: cnn_output_length(
        x, kernel_size, conv_border_mode, conv_stride)
    print(model.summary())
    return model

model_end = final_model(input_dim=13, # change to 13 if you would like to use
MFCC features
                        filters=200,
                        kernel_size=11,
                        conv_stride=2,
                        conv_border_mode='valid',
                        dropout= 0.2,
```

```
                     units=200)
```

The submission includes a `sample_models.py` file with a completed `final_model` module containing a final architecture that is not identical to any of the previous architectures.

The submission includes a detailed description of how the final model architecture was designed.

Nice job. Your reasoning is sound here. Did the model perform as well as you thought it would? Why or why not?

DOWNLOAD PROJECT

RETURN TO PATH